HW2 - EL2700

Mustafa Al-Janabi
970101-5035
musaj@kth.se

Muhammad Zahid
951102-4730
mzmi@kth.se

Part 1: Inverted Pendulum Model

Q1 - We sample a grid with 10 and 5 points for x_1 and x_2, respectively. What effect does a higher number of points have? What about the computational complexity?

The 10 refers to the number of states we consider for the angles between $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The 5 is in the same range but for the second state which denotes the angular velocity. By increasing these numbers we increase the number of states explored by the DP algorithm. This might lead to a more accurate solutions, but in so doing the computation time suffers majorly.

Q2 - The grid is created with a small epsilon away of $\pi/2$. Why?

When theta reaches $\frac{\pi}{2}$ or $-\frac{\pi}{2}$, the system losses reachability. By then, there is nothing the controller can do to bring the pendulum back to balance. So to avoid that a small epsilon as added.

Q3 - Motivate and explain how the problem is solved in your own words.

Finding the solution to the DP problem is done mainly in solve grid. Firstly, it looks for all time steps over the time-horizon. For every time steps we search all the states in the state-space and calculate a cost associated with that state as well as a optimal control signal u. These calculations are handled by the function *solve_dp*.

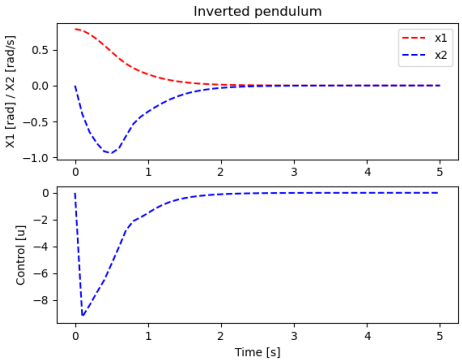
We then update the big matrices of U and J which contain all the calculated control inputs and costs for all states and time-steps. Lastly, based on the most recent cost that was calculated for all stated, the cost-to-go is updated. To accumulate for states that the systems might but which haven't been defined in the state space of the DP solver, the interpolation function is used to estimate the cost (and hence the optimal control) at those states too.

Q4 - Why do we include constraints on the state transition? Can this have undesired side-effects?

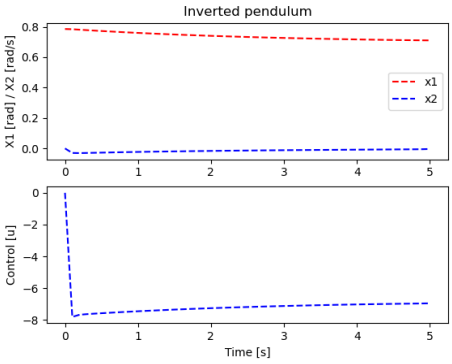
The constraints are added to the input u to prevent excessive control. Similarly, constraints are added on x so that it does not exceed the specified state-space and so that the DP problem is solved in finite time. The side effects to adding these constraints is that the DP solver might in some cases find optimal control signals which take the system outside the set boundaries. This typically happens at the state near the boundaries. This becomes evident when the debugger is enabled and the printing level of the *ipopt* solver is set to 3. Then, the solver claims that it has "converged on a local point of infeasibility". This, happens at the boundary.

Q5 - Tune the matrices Q and R with different weights in the task2.py script. Comment on the implication of different gains in the performance of the system.

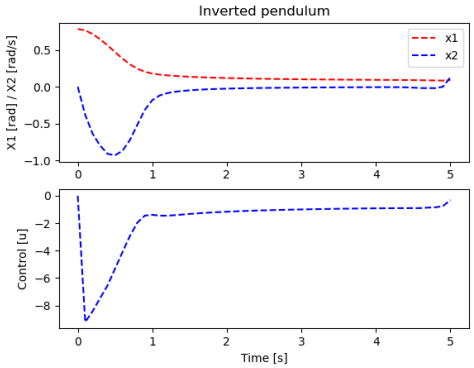
The matrix Q sets the weight for the state vector x and R sets the weight for the input u. When we penalize x_2 more by increasing Q_{11} to a large value, the controller does not try to bring the angle to zero. It balances the pendulum, so the angular speed remains almost zero. This can be seen in Fig. 1b.



(a) $Q[0,0] = 3 \times 10^9$



(b) $Q[1,1] = 3 \times 10^9$



(c) $R = 3 \times 10^9$

Figure 1: The state transition of the pendulum from the initial point $[\frac{\pi}{4}, 0]$

In a sense this penalizes any angular velocity in the system. On the other hand, as shown in Fig. 1c, when we increase R, the controller tried to minimize the control signal and, hence, the input energy required over the given time horizon. This could be seen as an approximation of the energy-optimal solution. Similarly, Fig. 1a shows that increasing $Q_0 0$ attempts to bring the angle close to zero as quickly as possible.

Nonlinear control design

By introducing a fictitious input v and updating the J_{stage} function, through feedback linearization. We get the true optimal input u^* as a function of the optimal v.

We have use the default values for Q and R, ie.

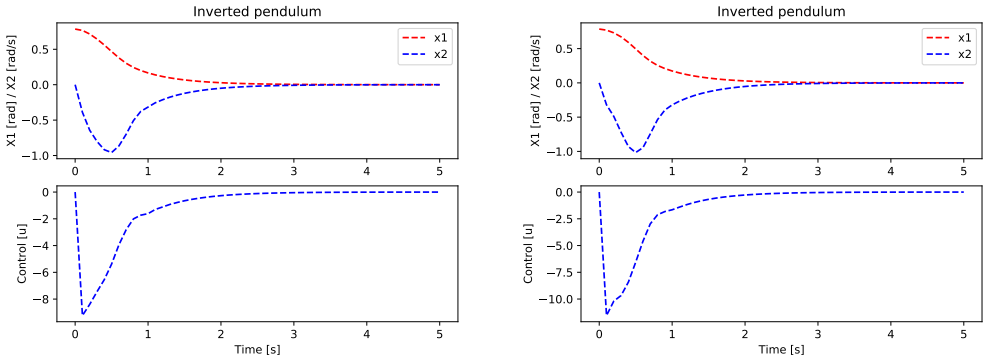
$$\begin{aligned} Q &= \begin{bmatrix} 3000 & 0 \\ 0 & 30 \end{bmatrix} \\ R &= 0.1 \end{aligned}$$

The comparison was done in two ways. Firstly, by applying the linear controller on the linearized discrete-time system and the nonlinear controller on the nonlinear system. Secondly, the controllers flipped so that we use the linear controller on the nonlinear system dynamics and vice versa. The results obtained are summarized in table 1. The computation time is calculated at each iteration and is then averaged over the total number of iterations. Furthermore, we note in Fig. 2 that time of convergence for both controllers is very similar, however, the nonlinear controller, applied on the nonlinear dynamics, has to expend more energy (with higher control), in order to achieve the same performance as the linear controller applied on the linear dynamic. On contrary, we note in Fig. 3 that the linear controller applied on the nonlinear dynamics struggles a lot, while the nonlinear controller on the linear system is very performant.

Table 1: Comparing the total accumulated cost, energy and computation time per time step expenditure of the controllers

	Cost	**Energy**	**Computation time [s/step]**
Linear controller on linear dynamics	9445	322	0.465
Nonlinear controller on linear dynamics	7485	280	0.485
Linear controller on nonlinear dynamics	2.3×10^6	3215	0.478
Nonlinear controller on nonlinear dynamics	9995	492	0.482

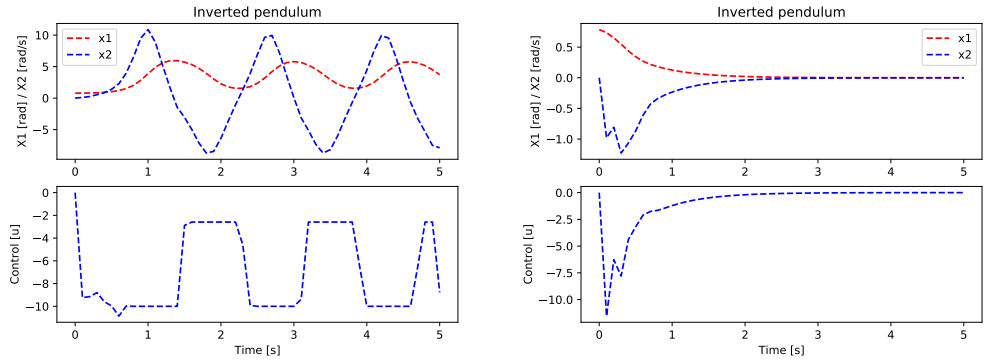
In conclusion, we see from the table above and the figures that the nonlinear controller handles both the nonlinear and the linearized dynamics of the pendulum system the best. The only downside is the slightly higher average computation time per time-step.



(a) The linear controller on the linearized system dynamics.

(b) The nonlinear controller on the nonlinear system dynamics.

Figure 2: Comparing the linear and nonlinear controllers applied on their respective environments



(a) The linear controller on the nonlinear system dynamics.

(b) The nonlinear controller on the linear system dynamics.

Figure 3: Comparing the linear and nonlinear controllers applied on each other's system dynamics.

Part 2: Cart-Pendulum model

In this section we extend our dynamical system to further include a cart that carries the pendulum. This leads to a system of four states, which proved to be a challenging control problem to solve using dynamic programming, in many aspects. In this section we will present the modifications we did to the gridder class *BiggerGridder* in order to account for the larger state-space, we then present the results we got and discuss some issues and challenges we faced.

Modifications of the gridder class

To begin with we had to define a new grid that takes into account the four states of the extended system. This was done through the following code in the `create_grid` function:

```
self.x1 = np.linspace(-2, 10, 5)
self.x2 = np.linspace(-3, 3, 5)

eps = np.pi / 10
self.x3 = np.linspace(-np.pi / 2 + eps, np.pi / 2 - eps, 5)
self.x4 = np.linspace(-np.pi / 2, np.pi / 2, 5)

self.U = np.zeros((len(self.x1), len(self.x2),
                   len(self.x3), len(self.x4),
                   int(self.sim_time / self.dt)))
self.J = np.zeros((len(self.x1), len(self.x2),
                   len(self.x3), len(self.x4),
                   int(self.sim_time / self.dt)))
```

Then the cost functions **J** are modified in `set_cost_functions` as follows

```
self.Jstage = ca.Function('Jstage',
              [x, Q, u, R],
              [(x.T - self.xr.T) @ Q @ (x - self.xr) + u.T @ R @ u])

self.Jtogo = ca.Function('Jtogo',
             [x],
             [(x.T - self.xr.T) @ self.Q @ (x - self.xr)])
```

Lastly, in solve grid we introduce five `for` loops that go through time and the grid of the four states as follows, in `solve_grid`:

```
for n in range(int(self.sim_time / self.dt)):
 for i in range(len(self.x1)):
  for j in range(len(self.x2)):
   for k in range(len(self.x3)):
    for l in range(len(self.x4)):
     x = np.array([self.x1[i], self.x2[j], self.x3[k], self.x4[l]])
     u, cost = self.solve_dp(x)
     self.U[i, j, k, l, n] = u
     self.J[i, j, k, l, n] = cost
```

Results

In Fig. 4 we see the obtained results for the controller optimized on the full system, with both the cart and the pendulum. Clearly, the code in functioning, however, the solution doesn't look satisfactory at all. The reference is set to $x = [10 \quad 0 \quad 0 \quad 0]^T$. As can be observed from Fig.4, the control action tries to move the cart, but soon the states go out of bounds and the input becomes zero. Many tuning attempts have been done in order to achieve more acceptable performance. No attempt gave the desired result that is comparative to the results of the state feedback based controller that was designed in the last homework. The challenges we faced and the reasons for this unsatisfactory results are discussed in the next section.

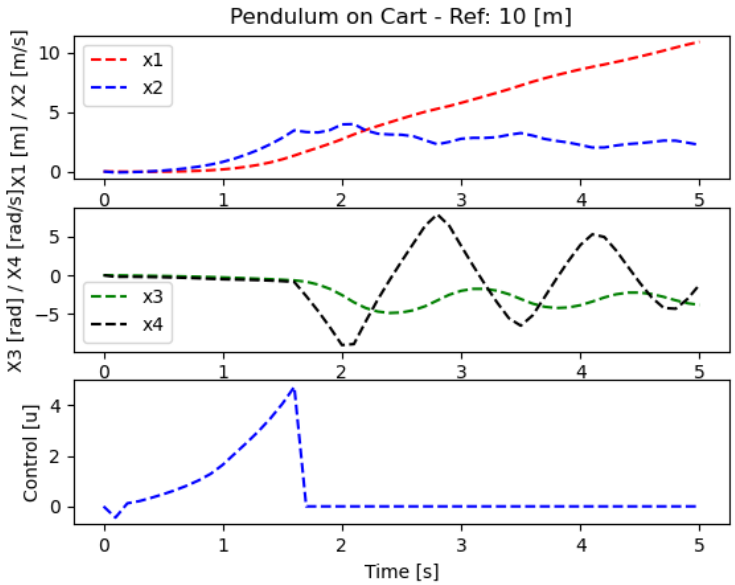


Figure 4: The performance of the four-state controller

Discussion

We started by enabling the Debug to TRUE, to get intermediate state transition information. We tuned the Q and R weights to restrict the velocities of the cart and pendulum, but that did not seem to give any better results. At the same time we also adjusted the state-space bounds and number of steps; although that did not seen to make any difference, so we stuck to few grid points to get faster results. Moreover, we played with the solver settings and tried different solvers; many solvers just simply gave us errors indicating that we had to make more adjustments to our problem formulation; hence, did not help. After enabling the solver prints we kept getting the following error:

```
EXIT: Converged to a point of local infeasibility.  Problem may be
infeasible.
EXIT: Optimal Solution Found.
```

So after a lot of tuning and adjustments, we were not able to find an optimal input sequence.